# Parallelizing Tabled Evaluations
## *Extended Abstract*

Juliana Freire [*], Rui Hu [†], Terrance Swift, David S. Warren [‡]
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400
Email: {*juliana,ruihu,tswift,warren*}*@cs.sunysb.edu*
Phone: (516) 632-8470

**Abstract:** SLG is a table-oriented resolution method that extends SLD evaluation in two ways. It computes the well-founded model for logic programs with negation with polynomial data complexity, and it terminates for programs with the bounded-term-size property. Furthermore SLG has an efficient sequential implementation for modularly stratified programs in the SLG-WAM of XSB. This paper addresses general issues involved in parallelizing tabled evaluations by introducing a model of shared-memory parallelism which we call *table parallelism* and by comparing it to traditional models of parallelizing SLD. A basic architecture for supporting table parallelism in the framework of the SLG-WAM is also presented, along with an algorithm for detecting termination of subcomputations.

**Keywords:** Parallel Logic Programming, Tabling, Table-parallelism, SLG, XSB.

## 1 Introduction

The deficiencies of SLD resolution are well known, and extended efforts have been made to remedy these deficiencies. For instance, while SLD can be combined efficiently with negation-by-failure in SLDNF, the semantics of SLDNF have proven unacceptable for many purposes, in particular for non-monotonic reasoning. Even without negation, SLD is susceptible to infinite loops and redundant subcomputations, making it unacceptable for deductive databases.

The latter deficiency, that of repeating subcomputations has given rise to many systems which table subcomputations: OLDT [18], and SLG-AL [19], and SLG [7, 6] are three tabling methods which have been implemented. At an abstract level, systems which use magic evaluation can be thought of tabling systems as well. Substantiation for this claim stems both from the asymptotic results of [14] and the experimental results of [17]. Tabling also appears to be relevant for computing the well-founded semantics: besides SLG, well-founded ordered search [15] and the tabulated resolution of [5] are two recent proposals which also use tabling.

Due to the power of tabling, many approaches have been formulated for its implementation — indeed [12] cites dozens. Nearly all of these approaches are sequential, however. We propose an abstract model, called *table-parallelism* for parallelizing tabled evaluations. In a sequential framework, SLG and SLD can be combined both theoretically and practically: the SLG-WAM evaluates SLD resolution with minimal overhead, and allows

[*]Partially supported by CAPES-Brazil.
[†]Supported by NSF grant CCR-9123200.
[‡]Partially supported by grants NSF CDA-9303181 and NSF CCR- 9102159.

free intermixture of SLD and SLG predicates. There is every reason to believe that when SLG is parallelized, it will be possible to mix table-parallelism with and- and or- SLD parallelism. While table-parallelism can exploit or-parallelism and some and-parallelism present in programs, it is orthogonal to both, as will become clear. In parallelizing practical programs, it is envisioned that a combination of all three approaches will prove beneficial, just as a combination of SLG and SLD works best for practical sequential programs.

The idea behind table-parallelism is simple: the table can be thought of as a large structured buffer, through which cooperating threads communicate.

**Example 1.1**
```
:- table a/2.
p(a,b).   p(a,d).   p(b,c).
q(b).   q(c).
a(X,Z):- a(X,Y),a(Y,Z).
a(X,Z):- p(X,Z),q(Z).
?-a(a,Z).
```

As a motivation for the parallel SLG model, consider the SLG forest in Figure 1 for the program of Example 1.1. Each SLG-subgoal can be thought of starting a new thread. Thus the call to a(b,Z) in node 5 begins a new thread, as does the call to a(c,Z) in node 11. Nodes like 0, 6, or 12, which use program clause resolution to produce answers, are called *generator* nodes, while nodes like 2, 3, 8, 9, or 14 which perform SLD resolution are called *interior* nodes.

Since each thread is itself an independent SLG-tree, each will have its own stacks, as well as its own heap and trail. The table space will be kept in shared-memory, as will the completion stack, kept for determining when to complete an SLG-subgoal. At a broad level, the parallel SLG-WAM differs from its sequential counterpart [16] in that it decouples the return of answers from the scheduling of their resolution, and in its use of a more complicated completion algorithm[1].

For in-memory queries, the SLG-WAM appears to be the fastest sequential tabling implementation currently available ([17]), and we believe its speed is due to its use of WAM-style Prolog compilation technology. This paper sketches a basic architecture for a shared-memory parallel SLG-WAM for definite programs — an architecture whose implementation is currently under development. We also discuss in detail how to detect when a tabled subcomputation has been completed. In a tabled evaluation, a given predicate may be part of a mutually dependent set of subgoals, called a *strongly connected component* or SCC[2]. The ability to dynamically determine both the membership of subgoals in an SCC and to detect when an SCC has been completed is necessary both to evaluate programs with negation and to efficiently evaluate definite programs. In a parallel framework, the detection of completion has much in common with termination detection for concurrent systems and we make use of results from concurrency theory in the proof of our algorithms correctness.

The rest of this paper is organized as follows. In Section 2 we describe the parallel execution model for SLG. The relations between table-parallelism and other forms of parallelism found in Prolog is discussed in Section 3. The issue of completion detection is addressed in Section 4, and in Section 5 the implementation framework of the parallel SLG-WAM is fully presented.

# 2 Table-Parallelism

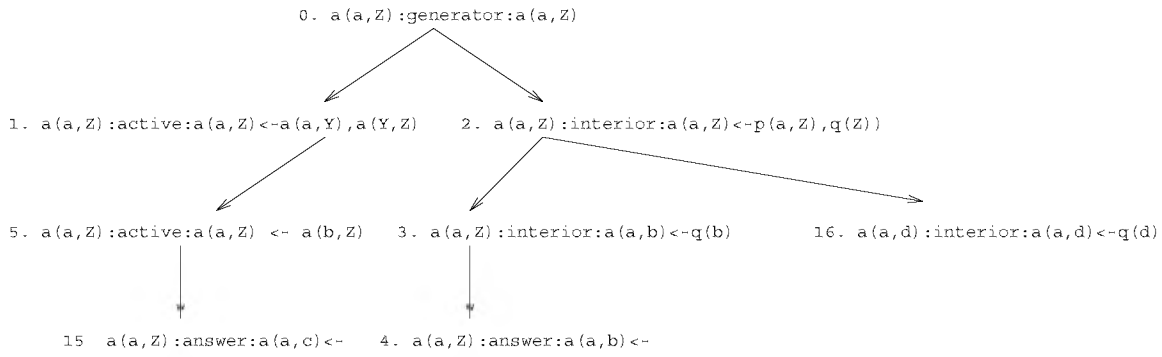
Tabling systems in general — be they SLG, OLDT, Magic or anything else — have four types of actions not found in SLD evaluation.

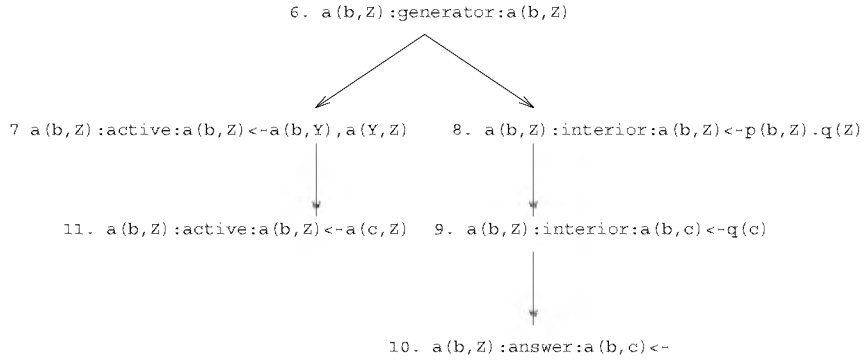
1. Creation of a new tabled subgoal;
2. Addition of an answer to a table;
3. Consumption of an answer from a table;
4. Determining when a set of subgoals is completely evaluated.

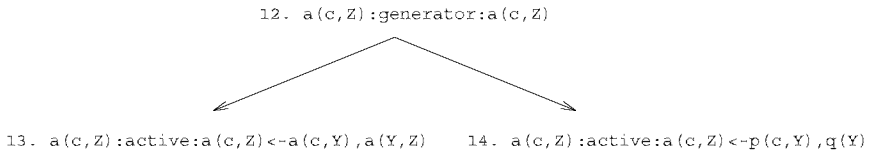
While the details of parallelizing these operations in a WAM framework are presented in Sections 4 and 5, the fundamental idea behind table-parallelism is to parallelize at creation of new tabled subgoals. (Or, as a practical matter, at creation of certain tabled subgoals). New threads for tabled subgoals are rooted in generator nodes and called *generator* threads: they generate answers and copy them into the table. The originating

[1] The SLG-WAM as described in [16] uses a tuple-at-a-time strategy for resolving answers with subgoals. While this strategy is efficient for logic programming, an alternate semi-naive strategy, has also been implemented for executing calls to disk.

[2] The term *recursive component* is also sometimes used for SCCs.




| Subgoal | Answer |
|---|---|
| a(a,X) | a(a,b) |
| | a(a,c) |
| a(b,X) | a(b,c) |
| a(c,X) | |

Figure 1: *SLG forest*

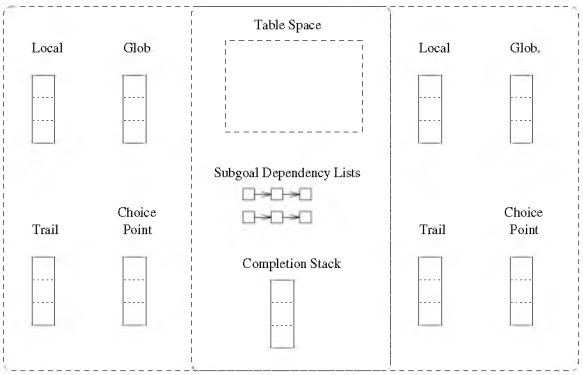



Figure 2: *Shared Memory Layout*

active node will asynchronously consume answers as they are added to the table, as noted in Section 1. Since communication is through the table, there are clear benefits to sharing the table. Likewise, the engine must dynamically detect when a set of subgoals is mutually dependent as well as when such a set has been completely evaluated. Thus, the *completion stack* which supports this decision is also kept in shared memory. In order to support efficient completion detection, a further structure, the subgoal dependency list will be kept in memory as well; its function will be explained below.

Memory layout for the parallel SLG-WAM is shown in Figure 2. Unlike many parallel Prologs neither the local stack nor the heap is shared: rather variable bindings are explicitly copied on subgoal call and answer return. Indeed, for tabling to work properly, a subgoal must be traversed *in its entirety* to determine whether it is new or not. Also, bindings of answers must in general be explicitly unified with the variables of subgoals. As a result, there is much less incentive to share WAM stacks in a tabling system than in a Prolog system[3].

Evaluations of the sequential SLG-WAM are deterministic, a property which the engine uses heavily. For instance, every time a new answer for a tabled subgoal is generated, it is scheduled on the choice point stack to be returned to the active subgoals that are variants of the tabled predicate. Properties of a depth-first evaluation are also used to detect of completion of an SCC. The first subgoal of an SCC visited by a sequential evaluation is called the *root* of the SCC. Whenever the system fails over the root of an SCC, it is provable that each subgoal inside the component is completed and all answers have been generated (Details can be found in [6]).

The strategy outlined above is unsuitable for parallel SLG, since active nodes for a particular tabled predicated might be in different threads. As a reflection of this difference, each thread will need to maintain a *subgoal dependency list* which keeps a pointer to the answers last resolved by each active node that corresponds to an uncompleted subgoal[4]. While the subgoal dependency list keeps thread-specific data, it is kept in a global area for use in the completion algorithm as will be discussed in Section 4.

The SLG-forest grows dynamically as predicates are called and thus the number of active threads depends on the particular program. For programs with high degree of table-parallelism, the number of active threads can be much larger than the number of actual processors. This would probably lead to a "slow down" if compared to the sequential execution, as most thread schedulers are slow at making general re-scheduling decisions [4]. In addition, most systems impose a limit on the number of threads that can be executing at a time.

As a first approach, we will require the user to decide which predicates to execute in parallel. Future research, will investigate scheduling strategies to distribute work among threads. For instance, a global queue may be kept so that whenever a tabled predicate is called for the first time, instead of spawning a new thread, the task would be added to the global queue. Available threads would then steal work from this queue. Trailing and scheduling mechanisms of the SLG-WAM already support the context switching required for this method.

One of the main goals of the parallel logic programming community is to exploit parallelism implicitly — including full Prolog with standard semantics [11, 10]. Prolog's implementation of SLD generates answer substitutions in textual order, but this notion does not exist in SLG, where the order of answer substitutions produced through clause resolution for generator nodes may not resemble the order produced by answer clause resolution with active (consuming) nodes. While SLD with cuts and SLG can be intermixed in the sequential framework, certain semantic changes to the cut are necessary to preserve the correctness of SLG. Parallelizing programs with the SLG cut remains an open issue. Aside from that,

[3] The SLG-WAM described in [16] avoids the reunification of answer clauses in certain, but not all cases.

[4] An optimization is available for completed subgoals.

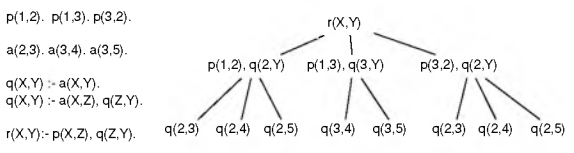



Figure 3: *Join in Parallel Prolog*

support for full Prolog can be provided by using standard techniques adopted by current or-parallel systems such as Muse [1] and Aurora [11].

A major advantage of parallel SLG over other parallel Prolog systems is the avoidance of redundant computation. In Figure 3, for instance, any or-parallel Prolog would have to compute the relation $q(2, Y)$ twice. In SLG, if the predicate $q$ were declared as tabled it would be computed only once, and any other subsequent call would simply consult the table. This avoidance of redundant computation has long been recognized as necessary for data-oriented queries. No less important is that SLG terminates for programs of bounded term size.

The copying of a subgoal each time a thread is created was a drawback of the Abstract Model [20]. Tabling systems, however, require copying of subgoals whether they are sequential or not[5]. When used judiciously, tabling may greatly improve performance over SLD ([3, 17]), and copying does not add a particular overhead for the parallel version. An advantage of using a copy-based method is that each thread can work independently on physically separated data, and issues related to environment sharing are avoided. (This property brings up the possibility of implementing parallel SLG on distributed memory machines).

# 3 Table vs. And/Or Parallelism

Because SLG is a different resolution method than SLD, the idea of parallelism in the context of SLG departs from traditional and- and or-parallelism. Parallelism will occur whenever a (parallel) tabled predicate is executed — not in the parallel execution of subgoals in the body of a clause, nor in the parallel execution of multiple clauses

[5] Reducing copying has been investigated for tabling systems in many contexts. Like most tabling systems, the SLG-WAM uses an array of static and dynamic techniques to address these issues ([13]).

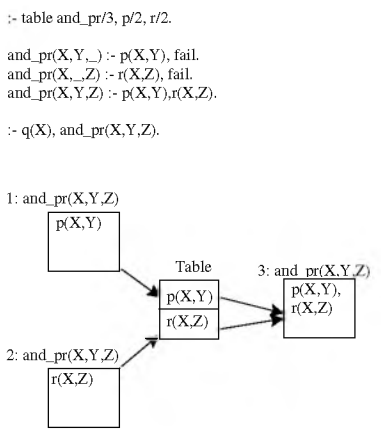



Figure 4: *And-parallelism in SLG*

defining a predicate.

The use of a copying strategy, and the lack of stack sharing between different threads allows each thread to work independently of the others on its own SLG tree in the SLG forest. Therefore, there is a great flexibility on the choice of evaluation strategy used to compute answers for the tabled predicate *within* each tree. The strategy could consist of sequential SLG, sequential SLD or any parallel model. In this sense, table-parallelism can be considered orthogonal to both and- and or-parallelism.

An interesting characteristic of table-parallelism is that by adopting an appropriate scheduling method, it may emulate or-parallelism. Given a predicate consisting of $N$ clauses, one needs only fold the body of each clause into a unique tabled predicate. If the subgoals are not present in the table, the caller spawns threads for each predicate in turn, failing — as a means of looking for new work — between each. (If the subgoals are present in the table, the consuming process reads them out, sequentially). We do not yet have any information about when such a strategy may be efficient. It is believed that table-parallelism may be most useful at coarser levels, or when redundant subcomputations are a problem.

Table-parallelism can also exploit some of the and-parallelism present in logic programs. Consider the goal `:- q(X), p(X,Y), r(X,Z)`. In parallel SLG, we could simulate and-parallelism by rewriting that goal as in Figure 4, and declaring `and_pr/3`, `p/2` and `r/2` as tabled. In this case all answers for `p/2` and `r/2` would be com-

puted in parallel, assuming that they are not already in the table. As they are being computed, they will be fed to the third clause of `and_pr/3`, where `p/2` and `r/2` will be active, (consumer) nodes.

A preprocessor could be easily written to generate code that emulates either behavior. If it were desired for table parallelism to emulate and- or or- parallelism, analysis could be performed to determine where a program contains and- and or- parallelism using traditional techniques. The output of the analysis phase would then be adapted as input to such a preprocessor. In practice, table-parallelism has its own strengths, which are mainly complimentary to traditional methods.

# 4 Parallel Completion

Detection of completion is a non-trivial problem in a sequential framework, and the difficulties are compounded for parallel evaluations. While various approaches to this problem are possible, implementation of a workable engine requires a minimum of synchronization: accordingly we base our approach on *distributed termination detection* algorithms as in [8].

Section 2 introduced the shared completion stack, used to detect completion of an SCC. The completion stack consists of frames for each incomplete tabled subgoal. The frames are pushed onto the stack when the subgoals are called, and consist of the following elements.

| | |
|---|---|
| *DFN* | Unique Depth-first number |
| *PosLink* | Earliest direct dependency |
| *PosMin* | Earliest indirect dependency |
| *StateFlag* | Thread is working |
| *AnswerFlag* | Thread has consumed all answers |
| *ColorFlag* | Thread may affect other subgoals |

$DFN$, $PosMin$, and $PosLink$ ([16, 6]) are used in the sequential model as well as the parallel, and together determine the extent of the SCC. For a given tabled subgoal $A$, $PosLink$ is originally set to the unique $DFN$ of $A$. If $A$ calls another incomplete tabled predicate $B$, whose $DFN$ is less than that of $PosLink(A)$, $Poslink(A)$ is set to $Poslink(B)$, so that $PosLink$ reflects the earliest direct dependency of $A$. Conversely, subgoals on the completion stack are assumed to depend on other subgoals of higher $DFN$. Consider the following situation. Subgoal $A$ calls a new subgoal $B$, and a new frame is placed on the completion stack. $B$ later calls a subgoal $C$ which is an ancestor of both. In this situation $A$ depends on $C$ through $B$ and none of these subgoals can be completed until they all are. We thus cannot complete $A$ before $C$, because $A$ depends on an incomplete subgoal $B$, younger than $A$, and $B$ depends on a subgoal $C$ older than $A$. In other words, $A$, $B$, and $C$ are in the same SCC, rooted at $C$. To prevent improper completion, information about the $PosLink$ values for these subgoals must be propagated to $A$ before it can be completed. The characteristics of this propagation vary from the sequential to the parallel model (See [6] for the sequential case). We let $PosMin(A)$ stand for the minimum $PosLink$ value of all subgoals younger than $A$ — information which is available from the completion stack.

The *StateFlag* indicates whether the thread is doing some computation, while the *AnswerFlag* indicates whether there are still answers for the thread to process. The *StateFlag* can have values *done*, *undone* or *unconditionally done*. Conceptually, the *AnswerFlag* can have values *done* or *undone*, however it is implemented as a pointer into the subgoal dependency list. The *AnswerFlag* for a thread $T$, will be a disjunction of the next answer pointer for all Active nodes in $T$. The *AnswerFlag* will be *done* when all answers have been used. In order to complete correctly, a computation needs to maintain the following invariants.

**Invariant 4.1 (StateFlag)** *The StateFlag is* done *for a given thread iff it is performing no answer or program clause resolution (i.e. doing no work).*

**Invariant 4.2 (AnswerFlag)** *The AnswerFlag is* done *for a given thread iff all active nodes in the SLG tree for the thread has resolved all applicable answers in the table.*

Whenever a thread exhausts all of its program and answer clauses it sets both *StateFlag* and *AnswerFlag* to *done*, and goes to sleep. Whenever a new answer is added to a table for any of the *Active nodes* in the thread, the thread's *AnswerFlag* is effectively set to *undone*, since the *AnswerFlag* is a list of pointers into the table. After a thread wakes up, it executes the following steps.

1. The thread set its *StateFlag* to *undone*.
2. The thread traverses its subgoal dependency list to see if new answers have been returned for its active nodes.

3. If new answers have been returned, the thread lays down an `answer_return` choice point (presented in Section 5) for each uncompleted subgoal. The engine will backtrack through the choice points and continue its normal execution. The engine iterates steps 1 and 2 while there is work to do.

4. When there is no more work left, the thread sets *StateFlag* to done (*AnswerFlag* has been effectively set to *done*) and sleeps.

Under the formalism of [6] or [16], the following proposition can easily be proven.

**Proposition 4.1 (Parallel Completion)**
*Let $\mathcal{E}$ be a computation in which Invariants 4.1 and 4.2 hold. Let*

$$DFN(S) = PosLink(S) = min(PosLink(A))$$

*for all A younger than S on the completion stack. Also let*

$$StateFlag(S') = AnswerFlag(S') = done,$$

*where $S'$ denotes S and every A younger than S on the completion stack. Then the subgoals from S to the top of the stack constitute a maximal SCC and have been completely evaluated.*

The intuition behind Proposition 4.1 is that if the *PosLink* values are as specified, the subgoal forms the root (oldest subgoal) of an SCC. If the flag values are as specified, and if the invariants hold for the computation, then all mutually dependent subgoals have performed all answer and program clause resolution. Proving correctness of completion then devolves upon ensuring that the invariants hold.

As an example of how the invariants can fail to hold, consider the following situation. We make use here, and throughout the paper of a convention that the root thread (the thread for the root subgoal of an SCC) leads the check for completion of the SCC.

Let *root* be the root of $SCC_{root}$, which is currently being checked for completion. And, suppose $SCC_{root} = \{r_{n-1}, r_{n-2}, ..., r_1, root\}$. *root* will check the status of the flags of each other subgoal, from $r_{n-1}$ to $r_0$, in this order. Suppose while *root* is checking the flags for $r_i$, after finding that all flags for $r_l$, $l > i$ are *done*, another

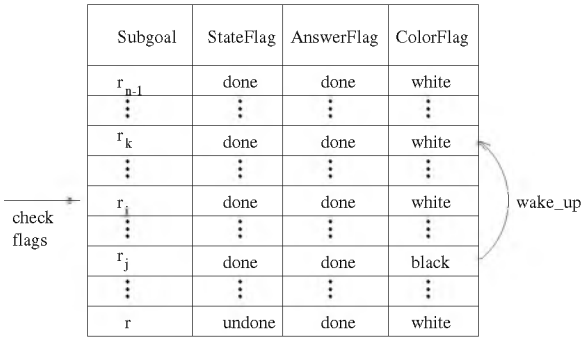



Figure 5: *Snapshot of Completion Stack*

thread $r_j$, where $0 \leq j < i$, generates a new answer and activates a thread $r_k$ that has been already checked by *root*. The root will wrongly assume that $r_k$ is done, and it will incorrectly complete.

Following the model of [8], we address this problem by introducing another flag, the *ColorFlag*, for each tabled subgoal.

*Definition 4.1 (ColorFlag)* The *ColorFlag* indicates that a node in the SCC may have generated new answers after another node was checked as completed by the root. It can be either *black*, if new answers were added, or *white*, otherwise. □

Initially the *ColorFlags* for all tabled subgoals are set to *white*. When after waking up, a thread $r$ generates some new answers it will set its own color to *black*. During the completion check, if the root finds a black node, it will know that some node, that it had already checked, might have been awakened (see Figure 5), and therefore, completion has to be restarted.

# 5 Implementation Framework

This section presents the instruction- level changes needed to parallelize the SLG-WAM. While SLG-WAM instructions for definite programs have been presented in [16], a detailed knowledge of these instructions is not needed to for understanding this section, since the changes mainly involve adding concurrency features to tabling operations. The changes are as follows:

- *When calling a tabled subgoal* — in the NEW ACTIVE instruction. If the selected subgoal is not al-

ready in the table, add an entry for it, create a thread and copy the subgoal into the thread. Update the calling thread's subgoal dependency list with a pointer to the active node. Then, `fail` in order to find other work. The copy of the subgoal is required so that variables are not shared between threads. On the other hand if the selected subgoal is already in the table but not complete, the *PosLink* value for the root subgoal should be updated, along with the subgoal dependency list. Resolution can then begin against answers in the table. Finally, if the subgoal is in the table and completed, the answers are treated as if they were asserted code. Neither the completion stack nor the subgoal dependency list needs to be updated.

- *When adding an answer for a tabled subgoal* — in the NEW ANSWER instruction. Besides adding an answer to the table, (if the answer is not already there), the instruction will set its *ColorFlag* to black. The *AnswerFlag*, which is a (pointer to a) list of pointers into the table, will be effectively updated as discussed in Section 4.

- *When checking for completion* — in the COMPLETION instruction. The thread must first check that no answers have been added that might unify with any of its active subgoals. If answers have been added the process schedules their return on the choice point stack. Otherwise, the predicate marks its *StateFlag* as *done* and, if the thread is the root of its SCC, it begins the completion check algorithm to be described below.

This model requires synchronization between threads when subgoals are created and when they are completed, but not otherwise. Adding a new SLG subgoal to the table will require a lock to prevent multiple threads from simultaneously executing the NEW ACTIVE instruction for the subgoal. In the SLG-WAM the table for each predicate is structured as a tree, so that only a subtree need be locked, allowing a great deal of concurrency. Contention will be more likely to arise in the completion stack. For adding answers to the table, however, no locking will be necessary as there is a single generating thread for each subgoal (although possibly multiple consuming threads).

The issue of returning new answers to existing subgoals differs from the tuple-at-a-time method described in [16], but resembles that of a sequential semi-naive implementation (to be described fully in a forthcoming paper). In the semi-naive implementation, an iteration consists of all work that occurs between failures back to the COMPLETION instruction for the root of an SCC. At iteration $i$, the engine uses answers from iteration $i-1$ to determine a new set of answers. At the end of each iteration the COMPLETION instruction checks whether new answers have been added in iteration $i$ or whether the SCC has instead reached *fixpoint*[6]. The semi-naive strategy can be incorporated into parallel execution: at each COMPLETION the engine returns any new answers added by other threads. Only in the case where no answers have been added and there is no other work to do, does the parallel algorithm differ from sequential semi-naive. Rather than being able to determine fixpoint, the consuming thread must coordinate with other threads to determine if all answers have been derived for its active subgoals. If the thread is a root thread, it checks the flags of the completion stack using an algorithm described below. Otherwise, the thread must poll at intervals to see whether new answers have been added.

Returning answers is summarized in the procedure *Iterate* in Figure 6. The subgoal dependency list maintains information for subgoals which are active for each thread. In order to make use of new answers at the end of an iteration, the thread will exploit two properties for an answer trie. A path from the root of the trie to a leaf node constitutes an answer for the trie. The first property exploited is that each leaf of the trie is linked together in a list for sequential access. When answers are added to a trie, the trie is extended, and the answers are also added to the end of the list, using a global *listend* pointer to the end of the trie. For each subgoal in the dependency list the thread creates *answer return choice points*, which contain, among other information, a local *listend* value for iteration $i$ and a local *listend* value for iteration $i-1$. The answer return choice point then backtracks through each active node for its subgoal and returns each answer in the *listend* interval to the node. Of course, there may be discrepancies between global and local listend values: a generator thread may add an answer as a consumer thread reads the *listend*. While the discrepancy is probably not important while a

[6] Failure back to the root of an SCC does not always reclaim space, since environments for SLG predicates are maintained between iterations.

```
Iterate
  If subgoal dependency list is empty
    AnswerFlag = StateFlag = uncond_done;
    update subgoal stack;
    exit;
  Else
    For each subgoal in list
      If (subgoal is unconditionally complete
      and there are no unexamined answers)
        Remove subgoal from list
      Else
        If there are unexamined answers
          last_listend = local_listend;
          local_listend = global_listend;
          create answer return choice point;
```

Figure 6: *Algorithm for iterating computation*

```
CheckCompletion(S)
  If Iterate exits
    exit;
  if S depends on a subgoal which is not complete
    if any choice point has been added
      StateFlag = AnswerFlag = undone;
      fail; {backtrack through the choicepoints}
    Else
      if S belongs to the youngest SCC
        StateFlag = AnswerFlag = done;
        if S is the leader
          for each r_i ∈ S's SCC such that r_i is
          not uncond_done
            wait ((r_i.StateFlag==done) and
            (r_i.StateFlag==done));
            if(r_i.ColorFlag==black)
              set the ColorFlags for all
              subgoals in the SCC to white;
              suspend and restart completion later;
          complete tables for all subgoals in SCC;
        Else
          sleep;
      Else fail; {not youngest SCC}
```

Figure 7: *Pseudo-code for* COMPLETION *instruction*

thread is in the process of iteration, it must be rectified by the time the subgoals are completed.

When a thread wakes up to poll for answers it will set its *StateFlag undone*. It will then run through its *subgoal dependency list* and return the new answers from the global table to the suspended nodes in its subtree and continue its normal, sequential SLG, execution. When it runs out of work, (or if there was no work to begin with) it resets its *StateFlag* to *done*. If in the course of its execution, this thread generates new answers, it will set its *ColorFlag* to *black*, indicating that other nodes might have been activated. Finally, if the current thread is not the leader of its SCC, it goes to sleep. Otherwise, it will start the completion check.

The COMPLETION instruction uses the algorithm in Figure 7. In the case where a thread depends only on subgoals which are unconditionally done, and where all answers have been examined, the thread marks its table as unconditionally complete and exits. Otherwise, if a thread traverses the subgoal dependency list and finds no new answers, its *AnswerFlag* is effectively done, so it sets its *StateFlag* to *done* in the completion stack (Since we assume only one thread per subgoal there will be no confusion). If a new answer is added for a subgoal upon which other subgoals depend, the subgoal dependency lists and *AnswerFlag*s will reflect this change. The thread generating a new answer also resets its *ColorFlag* to *black*.

During completion check the leader will check all subgoals in its SCC, that are not unconditionally done. If any subgoal has either *AnswerFlag* or *StateFlag undone*, the leader will wait until both are set to *done*. If both flags are *done*, the leader will check the *ColorFlag*. If the *ColorFlag* is *white*, the leader will proceed to the next subgoal. Whenever the leader reaches its own frame in the completion stack, by Theorem A.1, all subgoals in the SCC are completely evaluated, and its is safe to mark the tables complete, pop the completion stack, and terminate. After the root subgoal, or leader, completes all tables, it notifies the threads that it is safe to exit and exits itself. Otherwise, if the leader finds some *black* subgoal, a subgoal may have generated new answers that have not been consumed. The leader then fails and completion check will be restarted later.

The root thread thus enters a loop where it traverses the subgoal dependency list and either returns answers

or checks for completion, while the non-root threads either return answers or mark themselves as conditionally complete, by setting its flags to *done*. Since a given SCC may grow as new dependencies are added, different threads may become root at different times. If invariants 1 and 2 hold, the algorithm handles this situation. Suppose that thread $T_1$ is root and thread $T_2$ later becomes root. $T_1$ would only complete after it was determined that there were no unresolved answer or program clauses in the SCC. On the other hand, $T_2$ would have been made root only if some subgoal in the SCC rooted at $T_1$ discovered a dependency on $T_2$. But this cannot occur after all answer and program clauses have been resolved.

By formalizing the algorithm outlined in this section, it is provable that:

- *Threads terminate iff their subgoals have been completely evaluated;*
- *There will be no deadlock or starvation.*

The formal statement of these results and their proof are contained in the Appendix.

# 6 Conclusion

Tabling adds the ability to evaluate programs according to the well-founded semantics; it adds important termination properties to evaluations, whether or not the program contains negation; and it computes queries with polynomial data complexity, again whether or not negation is present in the program that evaluates the query. Depending on one's point of view, tabling is thus a desirable addition or a necessary addition to an evaluation. This paper constitutes a first step in the design of a parallel SLG-WAM for definite programs. Its implementation intended to be a continuation of the semi-naive sequential model, as indicated in Section 5. While implementation results are not yet available, the foregoing discussion indicates that only instructions for tabled predicates are to be changed by the design. As a result, for sequential SLD the times for the parallel engine should equal the sequential SLD times for the sequential engine: about a 10-15% overhead added to the WAM.

While we have introduced the model of table-parallelism, due to space limitations, we have only discussed its most basic model. As a first instance, the model parallelizes all tabled evaluation — clearly this is not always a wise strategy. A clear first extension would allow the user to parallelize only certain tabled predicates. As a second extension, SCCs could be analyzed statically and threads matched to SCCs rather than to subgoals. A third extension might be to make the completion stack into a tree, so that SCCs could complete early. While this last extension might cause some overhead in determining the root of the SCC, it would be expected to be reduce the synchronization needed for completion. These three extensions are straightforward, but their worth can only be ascertained in the context of an implemented system for which work is now underway.

**Acknowledgements**: We are indebted to Phil Lewis for useful discussions on the concurrency issues of the completion algorithm.

# References

[1] K.M. Ali and R. Karlsson. The Muse or-parallel Prolog model and its performance. In *Proceedings of the North American Conference on Logic Programming*, pages 757–776. MIT Press, 1990.

[2] Krzysztof R. Apt. Correctness proofs of distributed termination algorithms. *ACM Transactions on Programming Languages and Systems*, 8(3):388 – 405, July 1986.

[3] F. Banchilhon and R. Ramakrishnan. An amateur's introdution to recursive query processing strategies. In *Proc. of SIGMOD 1986 Conf.*, pages 16–52. ACM, 1986.

[4] A.D. Birrel. An introduction to programming with threads. Technical report, Digital - Systems Research Center, Palo Alto, California, 1989.

[5] R. Bol and L. Degerstedt. Tabulated resolution for the well-founded semantics. In *Proc. ILPS'93 Workshop on Programming with Logic Databases.* MIT Press, 1993.

[6] W. Chen, T. Swift, and D.S. Warren. Efficient implementation of general logical queries. *J. Logic Programming.* To Appear.

[7] W. Chen and D. S. Warren. Query evaluation under the well-founded semantics. In *Proc. of 12th PODS*, 1993.

[8] E.W. Dijks tra, W.H.J Feijen, and A.J.M. van Gasteren. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, pages 217 – 219, June 1983.

[9] Nissim Francez. *Program Verification.* Addison-Wesley, 1992.

[10] Gupta,G., Ali,K.M., Carlsson,M., and M. Hermenegildo. Parallel Logic Programming: A Survey. Unpublished manuscript, 55 pages. Available by anonymous ftp at ftp.cs.nmsu.edu.

[11] E. Lusk, R. Butler, T. Disz, R. Olson, R. Overbeek, D.H.D. Warren, A. Calderwood, P. Szeredi, S. Haridi, P. Brand, M. Carlsson, A. Ciepielewski, and B. Hausman. The Aurora or-parallel Prolog system. *New Generation Computing*, (7):243–271, 1990.

[12] R. Ramakrishnan and J. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 1994. To appear.

[13] P. Rao, I.V. Ramakrishnan, T. Swift, and D.S. Warren. Dynamic argument reduction for in-memory queries. In *2nd ICLP Workshop on Logic Programming and Deductive Databases*, 1994.

[14] H. Seki. On the power of alexandrer templates. In *Proc. of 8th PODS*, pages 150–159. ACM, 1989.

[15] P. Stuckey and S. Sudarshan. Well-founded ordered search. In *13th conference on Foundations of Software Technology and Theoretical Computer Science*, pages 161–172, 1993.

[16] T. Swift and D. S. Warren. An abstract machine for SLG resolution: definite programs. In *Proceedings of the Symposium on Logic Programming*, 1994. To Appear.

[17] T. Swift and D. S. Warren. Analysis of sequential SLG evaluation. In *Proceedings of the Symposium on Logic Programming*, 1994. To Appear.

[18] H. Tamaki and T. Sato. OLDT resolution with tabulation. In *Third Int'l Conf. on Logic Programming*, pages 84–98, 1986.

[19] L. Vieille. Recursive query processing: The power of logic. *Theoretical Computer Science*, 69:1–53, 1989.

[20] D.H.D. Warren. Or-parallel models of Prolog. In *Proceedings of the International Conference on Theory and Practice of Software Development.* Springer-Verlag, 1987.

# A Appendix: Correctness of Completion Algorithm

**Theorem A.1** *(Completion Algorithm) Given a set $S$ of mutually dependent tabled subgoals, the completion algorithm of Figure 8 will mark as complete the subgoals in $S$ iff $\forall A \in S$, $A$ is completely evaluated. Also, it is guaranteed that there will be no deadlocks or starvation.*

**Proof:** In order to prove Theorem A.1, we need to prove the following:

1. The completion algorithm will mark as complete the subgoals in $S$ iff $\forall A \in S$, $A$ is completely evaluated.
2. Deadlock is prevented. In other words, it will always be the case that either completion detection or subgoal computation can occur.
3. There will no be starvation. In other words, during completion detection, if the subgoals in the SCC are *not* completely evaluated, subgoal computation will resume.

Program verification techniques, such as the ones described in [9, 2], are needed for formal proof of the above four properties. The proof here is based on the distributed completion proof presented in [8].

When synchronizing threads for completion we may abstractly assume that the threads of the SCCs enter one of three procedures: *wakeup* to look for new answers; *completion* to check for completion of the nodes in the SCC; or *terminate.*

Using the structure of Section 5, the **wakeup** choice will be executed each time a tabled predicate starts to

```
wakeup:
   r.StateFlag = undone;
   reads the tables of all active subgoals in its tree;
   r.AnswerFlag = done;
   if (r generates new answers)
      writes the answers into r's table;
      r.color = black;
   r.StateFlag = done;
   sleep;
terminate:
   exit;
completion:
   if subgoal is not the root
      exit;
   for (i=n-1; i≥0; i--)
      while ((r_i.StateFlag==undone)
         or (r_i.AnswerFlag == undone)) ;
      if (r_i.color == black)
         for (i=n-1; i≥0; i--)
            r_i.color = white;
            suspend and restart completion later;
   broadcast to r_{n-1},...,r_1 to terminate;
   exit;
```

Figure 8: *Non-deterministic choices for tabled predicates.*

execute the **check_complete** instruction. At this point it will check whether there are new answers to the active subgoals in its SLG tree, and if that is the case, return the answers to these subgoals. If, while executing, this tabled predicate itself generates new answers, it will set its *ColorFlag* to *black*, since it might have awakened some other tabled subgoals in its SCC.

Only the root (leader) of an SCC will execute the **completion** choice. During **completion**, the root will check whether all subgoals in the SCC have completed, and if that is the case, it will complete the tables, signal the other threads to exit, and terminate. If some subgoal in the SCC is either executing program or answer clauses, the root will wait until this subgoal is done to resume the checking. If, while scanning the completion stack the root finds any subgoal whose *ColorFlag* is *black*, it whitens the *ColorFlags* for all subgoals in its SCC and suspends the completion check.

While executing the **completion** alternative, the root of the SCC, *root* ($r_0$ in Figure 8) checks the status of threads $r_{n-1}, ..., r_1, root$, in this order. The necessary condition for the completion of *root*'s SCC is that *root* succeeds the checking for all $r_i$. Let $t$ denote the number of the thread whose flags are currently being checked by the root, then completion checking ends with $t = 0$.

The system's state will be captured by an invariant, which we call $P$. In the sequel, $P$ will be constructed in a number of steps, each step consisting of an extension of the state space considered and of an appropriate adjustment of $P$.

As a first step, we solve the problem in the absence of wakeups: in that case an *undone* (active) thread can become *done* (passive), but a *done* thread cannot become *undone*. The conclusion that all threads are *done* has to follow from (i) the invariant ($P_0$ as shown below); (ii) $t = 0$; and (iii) the fact that the flags of the *root* itself are *done*. Furthermore, the invariant ($P_0$) has to hold, independently of the value of $t$, when thread *root* has initiated the completion checking, i.e., when $t = n - 1$.

The above requirements are met by $P_0$:

$$P_0 : (\forall i : t < i < N : thread\ r_i\ is\ done)$$

We can now design the checking of *AnswerFlag* and *StateFlag* of each thread so as to keep $P_0$ invariant. Initiation of completion checking establishes $P_0$. Sequentially checking the remaining threads, however, decreases $t$ by 1, and may falsify $P_0$. Invariance of $P_0$ is achieved by adopting:

**Rule A.1** *The root waits until the flags for thread $r_{i+1}$ are done before its checks $r_i$'s StateFlag and AnswerFlag.*

In the *completion* alternative of Figure 8, Rule A.1 corresponds to the statements:

```
while((r_i.StateFlag == undone)
      or (r_i.AnswerFlag == undone));
```

When the flags of the other threads are *done*, the checking of flags will in due time be reached by *root*; when in addition thread *root* is *done*, termination can be concluded. The above discussion comprises the rules for checking flags.

Next to be taken into account is the possibility of wakeup: $P_0$ is falsified when thread $r_i$ with $t < i$ becomes *undone* on account of being waken up. Since only *undone* threads wake up other threads, we deduce that

the waking up that falsified $P_0$ has been initiated by a thread $r_j$ with $j \leq t$. In order to solve the situation we adopt the weaker invariant $P_0 \vee P_1$, such that any wakeup that may falsify $P_0$ establishes $P_1$.

To this end each thread is postulated to be either black or white (represented by the $ColorFlag$ of each subgoal). For $P_1$ we choose:

$$P_1 : (\exists j,\ 0 \leq j \leq t, \textit{such that thread } r_j \textit{ is black})$$

Wakeup is prevented from falsifying $P_0 \vee P_1$ by adopting the following rule:

**Rule A.2** *A thread wakes up another thread with a number higher than its own makes itself black* [7].

We have to verify, however, that the information available at thread *root* combined with the weaker $P_0 \vee P_1$ can still suffice to conclude termination. Since

$$(t = 0 \ \wedge \ \textit{thread root is white}) \Longrightarrow \ \neg\ P1$$

detection of termination has not been disabled.

With the possibility of a thread's having the color black, a new phenomenon has been introduced, viz. that of the unsuccessful completion checking: when a black thread is detected, the conclusion of termination cannot be drawn. In a first instance this problem can be tackled by adopting:

**Rule A.3** *After failing the completion checking, thread* root *reinitiates checking for completion.*

This corresponds to the statement

```
suspend and restart completion later;
```

in Figure 8.

Without the possibility of transitions from black to white, such a rechecking may not be successful. Therefore our next task is to assure that the whitenings do not falsify the invariant $P_0 \vee P_1$.

In view of the fact that initiating a check for completion establishes $P_0$, we can safely adopt:

**Rule A.4** *Thread root initiates a completion checking by making itself white and starting checking the status of thread* $r_{n-1}$.

[7] For consistency with Section 5, a relaxed Rule 1 is used in Figure 8. This relaxed rule effectively states that a thread waking up another thread will blacken itself.

The thread *root* can safely be whitened, and we turn to the other threads. Since whitening a thread can falsify only $P_1$, but does do so when that thread's number exceeds $t$, we can safely adopt:

**Rule A.5** *When it restarts completion checking, the root whitens all threads with number greater than* $t$.

Since the restart of completion checking changes the value of $t$ to $n-1$, which establishes $P_0$, we can further adopt:

**Rule A.6** *When it restarts the completion checking, the root whitens all threads from 0 to* $N-1$.

Rule A.6 assures the correctness of the entire `for` loop in the **completion** choice of Figure 8.

The above whitening protocols suffice to prove that when the algorithm terminates then all subgoals of the SCC are completed. An iteration of completion checking initiated after all subgoals are completely evaluated will end with all threads white and hence, a reiteration of the completion check is guaranteed to succeed.

The property that when all subgoals of the $SCC$ are completed then the completion algorithm will terminate derives from the fact that the root will eventually check for completion and it will restart completion later if it fails this time.

The absence of deadlock can be easily seen from the fact that the computation of subgoals (**wakeup** alternative of Figure 8) and the completion checking of the root (**completion** alternative of Figure 8) need not wait for each other.

In order to prove that there is no starvation, we note that if during completion detection, there are subgoals in the $SCC$ which are not completely evaluated, these subgoals will be awakened later. After a thread wakes up to poll for answers, it will run through its subgoal dependency list and return the new answers from the global table to the suspended nodes in its subtree.

□